Journal of Pure and Applied Mathematics: Advances and Applications
Volume 1, Number 1, 2009, Pages 21-46

A MATHEMATICAL MODELLING APPROACH FOR SOFTWARE TESTING OPTIMIZATION

CRESCENZIO GALLO and GIANCARLO DE STASIO

Dipartimento di Scienze Economiche
Matematiche e Statistiche
Università di Foggia
Largo Papa Giovanni Paolo II
1-71100 Foggia
Italy
e-mail: c.gallo@unifg.it

Abstract

Software errors can be a serious problem, because of possible damages (and related costs) and the burden of the needed corrections. Software testing, whose aim is to discover the errors in software products, requires a lot of resources and from it derives the overall quality, (i.e., reliability) of the software product. It is thus susceptible of optimization:, i.e., the best equilibrium between the number of tests to make and the global expected value of discovered errors has actually to be achieved. In fact, it is not economically feasible to proceed with testing over a given limit as well as to execute too few tests, running in the risk of having too heavy expenses because of too residual errors. In the present work we propose some models for software testing optimization, making use of an integer linear programming approach solved with a "branch & bound" algorithm.

2000 Mathematics Subject Classification: Primary 90C08, Secondary 68W99.

Keywords and phrases: software testing optimization, integer linear programming, branch and bound algorithm.

Received November 22, 2008

1. Introduction

The presence of errors in system and application software causes remarkable costs, both because of the damages provoked and the burden of the needed corrections. Testing, an activity whose aim is to discover the (inevitable) errors embedded in software products, requires a lot of resources and costs during the development and maintenance phases [16].

The overall quality of the software product, especially in critical application domains such as the medical one [37], is therefore mostly related to the presence (or absence) of errors, that is to reliability that can be defined as the probability that it works properly (without errors) within a given period of time [30], [34], [17], [29]. Even though noting that error is not an intrinsic characteristic of software (because of its being tied to use conditions and user expectations) it is undoubtable that testing is a decisive factor in attempting to discover as soon as possible, and so reduce, the presence of errors [12].

Testing takes up a lot of (money and time) resources against an error discovery expectation, and is thus susceptible of optimization [4], [5]:, i.e., the best equilibrium between the number of tests to make and the global expected value of discovered errors has actually to be achieved. In fact, it is not economically feasible to proceed with testing over a given limit as well as to execute too few tests, running in the risk of having too heavy expenses because of too residual errors [36].

Of course, given the intrinsic probabilistic nature of error discovery, the value of a test-case execution may be assigned only with probabilistic (or statistical) criteria, or with a-priori (even rough) estimates [25], [38].

2. Software Testing

Testing is an activity that receives in input the software product and outputs a quality report. Validation is performed against requirements (which expose the software functional nature, i.e., “what it does") and project (which expose the software logical nature, i.e., “how it does") specifications [30], [12], [36], [7].

2.1. Testing approaches

Philosophies upon which testing methods are based are essentially two: Path Testing and Functional Testing. Path testing aims to “exercise" program graph paths. Functional testing aims to verify functional requirements, depending on input-output software relationships [32], [14], [20], [39], [11], [23], [21], [22], [29].

Such approaches are enforced by several studies revealing a strong correlation between software complexity (given by its structural properties) and error characteristics (number, type, discovery and correction time).

The program structure influences the number of the possible paths, (e.g., a program with a backword goto in its execution flow has potentially infinite paths); see [35], [15], [6], [25]. Being, however, an exhaustive testing (of all the possible paths defined by program logic and functionality) impossible, one my want to select an optimal (in economic sense) test-case set.

2.2. Testing strategies

There are “static” testing strategies, which do not imply program execution. They substantially lead themselves to a deep code inspection and reach, however, preliminary and limited outcomes [12], [9], [18], [10], [13], [31], [2], [29]. There are several “dynamic" strategies (based upon program execution), from which the most significant seem to be [12], [20], [24], [19], [13], [31], [2], [29].

Path testing. It tends to execute every program graph path at least once; given the huge number of possible paths (in a nontrivial program) this strategy can be applied only with drastic reductions. Besides, it may be observered that the candidate errors to be revealed by path testing are those of logic and computational type.

Branch testing. It requires each branch in a program to be tested at least once; it is one of the simplest and best known strategies, but obviously less effective than path testing because error depends more upon branches combination than upon a single one.

Functional testing. It aims to verify software functionalities independently from its internal structure, (i.e., implementation logic). It requires test-case selection to be made upon functional relationships between input and output domain values. Being of course all the possible I/O relationships near to infinity, here too there is the problem of extracting a suitable test-case set.

Structured testing. It approximates path testing, and requires the software system under test to be broken down into a hierarchy of functional modules, in which the single modules are first tested (functional testing) and then their integration inside the whole software system is verified (integrated path testing). Structured testing increases branch testing reliability because it allows to verify branch combinations.

No one testing strategy is complete and reliable, however, in the sense that no one of them guarantees software correctness.

2.3. Approaches for testing optimization models

The behavior of tests effectiveness with respect to allocated resources (work, money, time, computation) is asymptotic (see fig.1). This means that, among all possible tests, only a part of them is economically useful [3], [36], [7], [9], [33], [8], [38].

Testing optimization is a particular resource allocation problem. Let us assign to each program's runnable path its test execution cost, (e.g., related to execution time) and its test execution value related to possible found errors, (e.g., related to instructions number and/or program blocks executed).

The tests' execution total value is the sum of selected paths' tests execution, and the tests' execution total cost is the sum of their corresponding costs.

Two possible approaches to the definition of models of testing optimization are:

- (M_1) **Limited resources model.** To maximize the total tests' execution value, given a maximum total execution cost.
- (M_2) **Given quality model.** To minimize the total tests' execution cost, given a minimum total execution value.

Figure 1. Test effectiveness related to resources.

3. Models for Testing Optimization

3.1. Model variables

Definition 1 (Program Block)**.** A program unit which consists of a set of consecutive instructions executed sequentially.

Definition 2 (Program Path)**.** A sequence of (not necessarily consecutive) repeatable blocks; the first block contains the starting program's instruction, and the last contains an "end program" instruction.

Definition 3 (Path Length)**.** The number of (repeatable) blocks in a program path.

Definition 4 (Simple Path)**.** A path without repeated blocks.

Definition 5 (Runnable Path)**.** A path for which exists some input data forcing its execution.

Subsets of runnable paths can be selected through the typical functional testing techniques, not requiring the construction of tests involving all possible combinations of input data values, but only those meaningful for program functionality.

Let us suppose to identify, inside a program, M blocks $b_i (i = 1 \dots M)$. Let N the number of runnable paths $p_j (j = 1 \dots N)$ determined through functional tests. Let:

Definition 6 (Composition Matrix)**.** *A* matrix $A = \{a_{i,j}\}$ defined as $a_{i,j} = 1$ if the block b_i belongs to the runnable path p_j, 0 otherwise (for $i = 1 \dots M$ and $j = 1 \dots N$). Its columns characterize the b_i blocks of which each p_j path is made up, while its rows characterize the p_j paths containing each b_i block.

Definition 7 (Coverage Degree)**.** The coverage degree of a runnable path is the ratio between the number of its distinct blocks and the number of all program blocks, i.e.,: $T_j = \frac{1}{M} \cdot \sum_{i=1}^{M} a_{i,j}$ $(j = 1 \dots N)$.

Definition 8 (Execution Matrix)**.** *A* matrix $E = \{e_{i,j}\}$, where $e_{i,j}$ represents the number of times the block b_i is executed within the runnable path $p_j (i = 1 \dots M;\ j = 1 \dots N)$.

Definition 9 (Block Length)**.** Let l_i the length of a block expressed in number of statements.

Let us finally associate to each runnable path p_j a bivalent integer variable $x_j = 1$ if p_j is selected for testing, 0 otherwise $(j = 1 \dots N)$.

3.2. The objective function and the constraints

Let c_j the test execution cost of p_j, and v_j its execution value. E.g., the cost can be initially considered proportional to only the execution time, and not being dependent on error debug and fix times. Test execution value can then be considered equal to the number of statements (code lines) involved by the control flow during p_j execution:

$$v_j = \sum_{i=1}^{M} e_{i,j} l_i \qquad (i = 1 \dots N).$$

To hold account of the number of blocks covered by each path p_j, we correct its value through the coverage degree T_j, so its execution test value becomes:

$$v_j = T_j \sum_{i=1}^{M} e_{i,j} l_i \quad (j = 1 \dots N). \tag{1}$$

The objective function z_1 of the testing optimization model M_1 (representing the total tests' execution value) is:

$$z_1(x) = \sum_{j=1}^{N} v_j x_j, \quad \text{to be maximized}. \tag{2}$$

For model M_2 the objective function, representing the total tests' execution cost, is:

$$z_2(x) = \sum_{j=1}^{N} c_j x_j, \quad \text{to be minimized}. \tag{3}$$

In the M_1 approach, the overall value's maximization must be done for a given maximum total test execution cost $c_{\max}$, i.e.,:

$$\sum_{j=1}^{N} c_j x_j \le c_{\max}, \; x_j = \{0, 1\} \; (j = 1 \dots N).$$

The M_2 model requires the total test execution cost to be minimized for a given minimum total test execution value $v_{\min}$, i.e.,:

$$\sum_{j=1}^{N} v_j x_j \ge v_{\min}, \; x_j = \{0, 1\} \; (j = 1 \dots N).$$

Both are integer bivalent linear programming models.

3.3. Block redundancy effect

The previous base models do not allow for program's topological coverage factor, definable as the ratio between the number of distinct

blocks within the chosen paths, and the total number of blocks in program.

In fact, because M_1's objective function and M_2's constraint does take into account the total number of executed - but not necessarily distinct - blocks, it may happen that the paths selected in an optimal solution of M_1 or M_2 cover a number of distinct blocks lower than those coverable within another nonoptimal (but obviously better) solution:, i.e., the redundancy effect of blocks within the various test-case paths (measuring the correlation degree between paths in terms of common blocks) is not considered.

Definition 10 (Redundancy Effect)**.** The "Redundancy Effect" can be defined, for each block b_i, as the number of paths in which the block is present respect to the total number of paths:

$$RE_i = \frac{1}{N}\sum_{j=1}^{N} a_{i,j}. \tag{4}$$

Definition 11 (Global Redundancy Effect)**.** The "Global Redundancy Effect" of blocks for all blocks is given by all RE_i's average value, i.e.,:

$$GREB = \frac{1}{M \cdot N}\sum_{i=1}^{M}\sum_{j=1}^{N} a_{i,j}. \tag{5}$$

3.4. Changing M_1 and M_2 models

In order to allow for block redundancy effect, a path execution test value may be corrected according to the value of the remaining selected paths and their blocks. This corresponds to associate to each block N partial values (one for each path), representative of the contribution given by the block to the test value of each path.

Let $w_{i,j}$ the partial value associated to block b_i in path p_j. Let $w_{i,j}$ be positive only for those paths it belongs to, i.e.,:

$$w_{i,j}\begin{cases} > 0 \text{ if block } b_i \text{ belongs to path } p_j, \\ = 0 \text{ otherwise,} \end{cases}$$

and the sum of the partial values of the blocks be the path's overall value v_j, i.e.,:

$$v_j = \sum_{i=1}^{M} w_{i,j} \quad (j = 1..N). \tag{6}$$

Then, if we compare the right members of (1) and (6) we obtain:

$$w_{i,j} = e_{i,j} l_i T_j \quad (i = 1 \dots M;\ j = 1 \dots N).$$

To obtain the best overall value, it will be necessary to hypothesize the inclusion, in the solution, of the blocks in those paths in which they have the best (maximum) partial value $w_{i,j}$; from that, the total test execution value allowing for the blocks' redundancy effect is given by the total sum of the maximum partial values of each block.

From above said, the new formulation of model M_1 becomes:

$$\max z_1(x) = \sum_{i=1}^{M} \max_{j=1\dots N} \{w_{i,j} x_j\}, \text{ subject to}$$

$$\sum_{j=1}^{N} c_j x_j \le c_{\max},\ x_j = \{0, 1\} \quad (j = 1 \dots N).$$

Model M_2 becomes:

$$\min z_2(x) = \sum_{j=1}^{N} c_j x_j, \text{ subject to}$$

$$\sum_{i=1}^{M} \max_{j=1\dots N} \{w_{i,j} x_j\} \ge v_{\min},\ x_j = \{0, 1\} \quad (j = 1 \dots N).$$

A completeness index of the test done is given, for both models, by the ratio between the obtained total test value and the desired total test value, i.e.,:

$$R = \frac{\sum_{i=1}^{M} \max_{j=1\dots N} \{w_{i,j} x_j\}}{\sum_{i=1}^{M} \max_{j=1\dots N} \{w_{i,j}\}}.$$

4. Solution Algorithm for the Proposed Optimization Model

The two proposed optimization models are similar; we develop model M_1 because it appears more suitable to real situations.

Being M_1 a bivalent integer linear programming model, the most efficient (from the application simplicity and convergence speed to optimum solution point of view) algorithms are those based on "branch & bound" techniques. In particular the Land and Doig algorithm [26] has been chosen, for the "knapsack" suitably modified problem. This algorithm has some valuable advantages, such as:

- it is simple to apply;
- the maximum number of steps for it to converge is about 2^N (where N is the number of p_j chosen runnable paths);
- it is of combinatorial type and then speedier than an enumerative-like algorithm.

Moving forward in algorithm's application, the initial problem is transformed into more and more simple ones, expressed similarly to the model, but with an ever lowering variables' number. Land and Doig's algorithm's formulation for the model's solution is:

$$\max z(x) = \sum_{i=1}^{M} \max_{j=1\ldots N} \{w_{i,j} x_j\}, \text{ subject to}$$

$$\sum_{j=1}^{N} c_j x_j \le c_{\max}, \; x_j = \{0, 1\} \quad (j = 1 \ldots N).$$

4.1. Initialization

Let M blocks constitute the program P to be tested, and let us suppose N runnable paths have been discovered (through the functional testing of P's specifications), among those a solution has to be chosen.

We may initialize the algorithm building the table T illustrated in fig. 2, in which the columns $1..N$ of V, W and C are sorted on descending

values of v_j / c_j. In this way it is possible to insert, in higher levels of the "branch & bound" algorithm's tree (and so evaluate as soon as possible) those paths having a higher probability to become part of the problem's optimal solution, so reducing the number of steps needed for the algorithm's convergence.

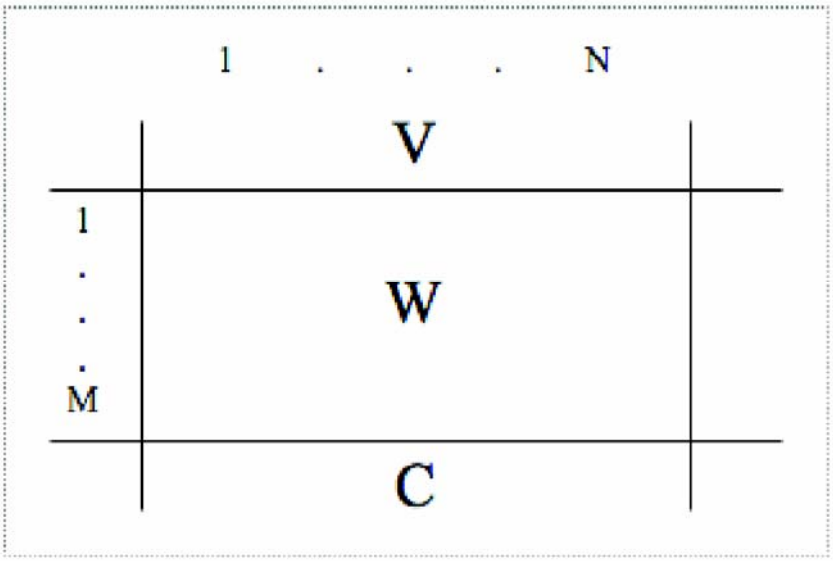

Figure 2. Algorithm's initialization table *T*.

4.1.1. Initial upper bound computation

Let $c_s = \min_{j=1..N}\{c_j\}$. Obviously, no more than $c_m = c_{\max} / c_s$ variables x_j can be equal to 1 (because of the first model's constraint); the initial upper bound is then given by:

$$L = \sum_{i=1}^{M} \max_{j=1..N}\{w_{i,j} x_j\}. \tag{7}$$

However, rather than c_m's value it is important to know whether $c_m \geq 1$, for which *L*'s value makes sense; otherwise, if $c_m < 1$, there is no feasible solution.

4.2. The separation

The separation is carried out through first giving to first variable on the left of table T the value 1 and then the value 0. So two (sub) problems are obtained: $\langle 1 \rangle$ and $\langle \tilde{1} \rangle$, formulated like the initial model and for which upper bounds for $z(x)$ will be computed.

4.3. Computation of the following bounds

Now we'll examine the possibility of inserting path 1 into the solution. Let $c_s = \min_{j=1..N}\{c_j\}$.

As for problem $\langle 1 \rangle$, i.e., the problem in which is $x_j = 1$, no more than $c_m = c_{\max}.c_1 / c_s$ variables $x_j (j = 2..N)$ can be 1. If $c_m \geq 1$ an upper bound $L\langle 1 \rangle$ for the value of $z(x)$ is given by: $L\langle 1 \rangle = v_1 +$ "total sum of maximum partial values $w_{i,j}$ relatively to blocks not belonging to path p_1"; if $c_m < 1$, the upper bound for $z(x)$ is only given by: $L\langle 1 \rangle = v_1$.

With regard to problem $\langle \tilde{1} \rangle$, instead, no more than $c_m = c_{\max} / c_s$ variables $x_j (j = 2..N)$ may have the value 1. If $c_{\tilde{m}} \geq 1$ an upper bound $L\langle \tilde{1} \rangle$ for $z(x)$ is given by the total sum of maximum partial values $w_{i,j}$, except for those related to blocks belonging to path p_1; if $c_{\tilde{m}} < 1$, no feasible solution exists starting from problem $\langle \tilde{1} \rangle$.

The representative tree is now the one given in Figure 3.

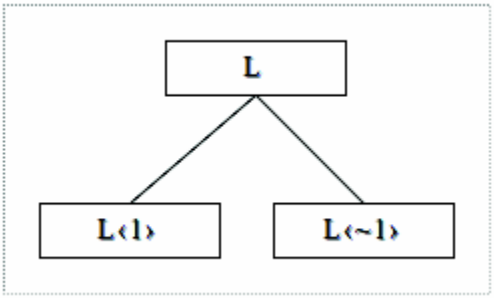

Figure 3. Representative tree-first step.

4.4. Choosing the separation vertex

We define strictly feasible a feasible solution to which a further path cannot be added without it becoming unfeasible. Let L_s the upper bound of $z(x)$ associated to the best strictly feasible solution. Initially, L_s must be set to zero.

As regards the pending vertex to which bound $L\langle 1\rangle$ is associated, it will be said active if $L\langle 1\rangle \geq L_s$, otherwise it will be exhausted. Analogously for $L\langle \widetilde{1}\rangle$.

If only one vertex is active, it will be selected for subsequent separation; otherwise, the following procedure will be applied. If $L\langle 1\rangle > L\langle \widetilde{1}\rangle$ separation will be done on the vertex to which bound $L\langle 1\rangle$ is connected by first giving the second x_j variable to the left in table T the value 1 and then the value 0. So, two new problems $\langle 1, 2\rangle$ and $\langle 1, \widetilde{2}\rangle$ are obtained, formulated like the initial problem and for which the related bounds $L\langle 1, 2\rangle$ and $L\langle 1, \widetilde{2}\rangle$ will be computed.

If instead $L\langle 1\rangle < L\langle 1, \widetilde{1}\rangle$, separation will be done on the vertex associated to bound $L\langle \widetilde{1}\rangle$, obtaining problems $\langle \widetilde{1}, 2\rangle$ and $\langle \widetilde{1}, \widetilde{2}\rangle$ for which the associated upper bounds $L\langle \widetilde{1}, 2\rangle$ and $L\langle \widetilde{1}, \widetilde{2}\rangle$ will be computed.

4.5. Generalizing the vertex's choice on which carrying out the separation (branch)

At a given step of the algorithm's application let us suppose to have "t" pending vertices and their related bounds $LP_1, \ldots, LP_t$. Let L_s the bound associated to the best strictly feasible solution (see 4.4).

A generic pending vertex r will be said active if $LP_r \geq L_s (r = 1..t)$; it will be said exhausted if $LP_r < L_s$. As already noted, the separation has to be done on active vertices.

Generally, there is more than one vertex in an intermediate step of the algorithm; among these, the active one with the maximum associated bound will be selected. If there is more than one "candidate" vertex for the separation, a random one will be picked out; in this case, one may agree to separate upon the candidate vertex representative of the most recently considered problem. The remaining vertices will surely be taken into account in the following steps.

Let us observe how such a generalization also holds for the first step of the algorithm, where the last pending vertices are those to which the upper bounds $L\langle 1\rangle$ and $L\langle\widetilde{1}\rangle$ are associated. Moreover, picking as separation candidate vertex the one with the maximum bound is a "trick" limiting (together with sorting table T's columns on descending v_j / c_j ratios) the number of problems to be examined, i.e., the number of steps in the algorithm. It is in fact useless to proceed on branching starting from an exhausted vertex, because only worse solutions than those already found would be obtained.

4.6. Generalizing $z(x)$'s upper bounds computation (bound)

Let the separation candidate vertex the one to which the problem indexed by $(\sharp 1, \sharp 2, \ldots, \sharp r)$, $r < N$ is associated, where $\sharp k = k$ if variable x_k has been set to 1, (i.e., if the path p_k has been selected), $\widetilde{k}$ otherwise.

Separation is done by first giving to variable x_{r+1} in table T the value 1, and then the value 0. Two new problems are so obtained, which will be denoted by $(\sharp 1, \sharp 2, \ldots, \sharp r, r+1)$ and $\left(\sharp 1, \sharp 2, \ldots, \sharp r, \widetilde{r+1}\right)$ formulated like the initial problem and for which the associated bounds will be computed.

For every new problem, the computation of $z(x)$'s upper bound will be executed as follows.

Besides the paths whose associated x_j variables have value 1 (for $j = 1..r+1$) it comes out that, letting $c_s = \min_{j=2..N}\{c_j\}$, no more than $c_m = (c_{\max} - \sum_{j=1}^{r+1} c_j x_j) / c_s$ variables can be set to 1.

If $c_m \geq 1$ an upper bound for $z(x)$ will be given by the total sum of the maximum partial values of each block, relatively to those belonging to paths for which $x_j = 1 (j = 1..r+1)$, plus the sum of the maximum partial values of each block, in solution, not yet covered by at least one path. Formally:

$$L(\sharp 1, \sharp 2, \ldots, \sharp r+1) = \sum_{i=1}^{M} \max_{j=1..r+1}\{w_{i,j} x_j\} + \sum_{i=1}^{M} \max^{\wedge}_{j=r+2..N} \{w_{i,j}\},$$

where $^{\wedge}$ means that in every ith iteration the maximum among the $w_{i,j}$ has to be computed only if the block b_i has not been selected in any path of the partial solution.

If $c_m < 1$, then we have only: $L(\sharp 1, \sharp 2, \ldots, \sharp r+1) = \sum_{i=1}^{M} \max_{j=1..r+1} \{w_{i,j} x_j\}$. Let us observe, too, that if $c_m \geq 0$ the partial solution is not feasible. Such a generalization holds also in the initial upper bound computation; in fact, we have (still being $x_j = 0$ for $j = 1..N$): $c_m = c_{\max} / c_s$ and then:

$$L = \sum_{i=1}^{M} \max^{\wedge}_{j=1..N}\{w_{i,j} x_j\} = \sum_{i=1}^{M} \max_{j=1..N}\{w_{i,j} x_j\}.$$

5. Application of the Model in Software Maintenance

In this section it will be examined the application of the proposed model of testing optimization to the software being modified in the maintenance phase.

5.1. Software maintenance

It has been estimated that software maintenance costs are in a ratio of 3 to 1 with the production ones; the maintenance activity, besides,

tends to deteriorate the software product's quality and reliability [30], [36], [28], [33], [18], [31], [27], [29].

The maintenance process objectives are [30], [14], [34], [17]:

- correction of errors missed in the testing phase and discovered during use;
- introduction of new functionalities;
- elimination of no more required functionalities;
- adaptation of software to new hardware configurations;
- possible optimization.

Whichever the objective is, the maintenance operation requires the following steps:

1. determining the detailed lists of the maintenance operation;
2. understanding of the software module(s) on which to act;
3. finding the acting points;
4. finding the elements upon which the adopted changes' effect spreads;
5. final testing of the changes.

Given the high algorithmic content and architectural complexity of the system software, one thinks that this is the more interesting area in the application of the techniques exposed in the present work. The more interesting is step 4 in which, after applying the required corrections to software after the specific operation, one passes to examine all the instructions (or sets of instructions, such as, e.g., blocks, routines, functional modules) which has to possibly be corrected because of the adopted changes.

Above all, step 5 is particularly important, in which we aim to apply the optimization model only to those paths being influenced by changes.

5.2. Inter-blocks influence matrix

Although in literature [38] attempts are present to automate step 4, (i.e., finding the software parts directly or indirectly influenced by changes) we think this becomes a very difficult task when a real (and thus nontrivial) application is concerned.

By the way, a valid help to this task may come from the "influence matrix" $IB = \{i_{k,p}\}$ among the blocks of a program, so defined [1]:

$$i_{k,p} = \begin{cases} 1 \text{ if block } k \text{ influences block } p \text{ (directly or not)} \\ 0 \text{ otherwise} \end{cases}$$

for $k, p = 1..M$, being M the number of distinct blocks in the program.

Because a block is made of a sequence of statements, we may say that there is a connection (or influence) between two blocks if they share some variables (created within the first and referenced within the second block) and are connected by the control flow both directly (in a single step) and indirectly (through several steps).

Let us assume that:

- if an array element is created or used (referenced), we agree that the entire array is created or referenced;
- as far as the interface variables of subprogram calls, we agree that those previously generated are referenced, while the others are created;
- in subprogram calls the global variables are there both created and referenced.

In figures 4 and 5 are represented a sample program and its related influence matrix *IB*, respectively. Taking account of the meaning of *IB*'s elements, its more immediate use regards the fast finding of the program blocks (or software system modules) to examine after a change.

In fact, let us observe that a change made on a program can whether involve the rewriting or deletion of existing statements, or the introduction of new instructions or, eventually, a combination of the previous.

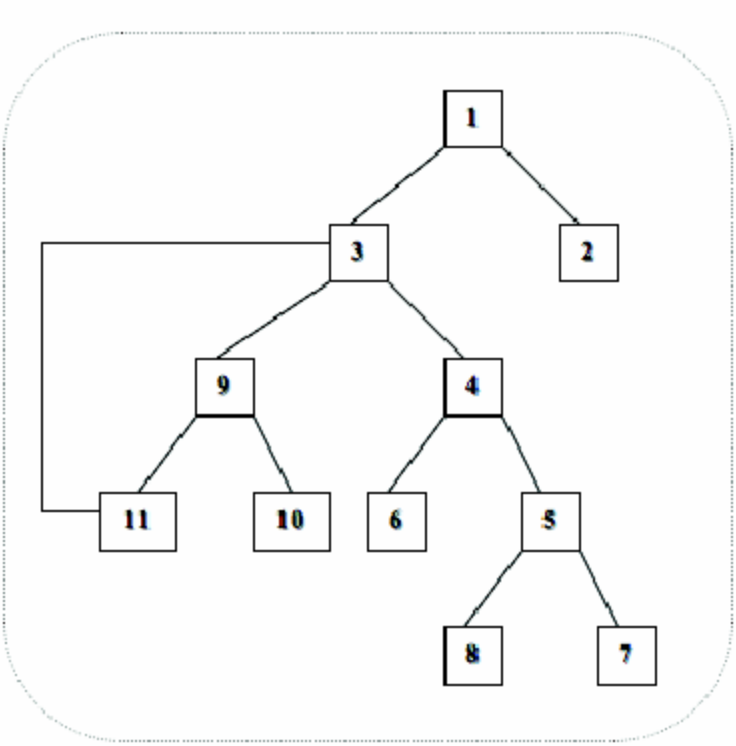

Figure 4. Sample program's graph.

Therefore, whichever is the operation to be completed upon block $k's$ instructions, it is obvious that it turns out necessary to examine, generally, all the blocks influenced by b_k; such blocks are simply found after the examination of the kth row of matrix *IB*.

In some cases (as for example when in the block k the control flow is altered or a new variable is created) the change "upsets" the entire influence network around b_k : it is thus necessary to verify all blocks influencing b_k, too. In this case it will suffice to examine also the kth column of matrix *IB*.

The examination of the new matrix *IB*, obtained after having applied all the necessary operations, moreover concurs to verify the congruence (with respect to the objectives of the maintenance process) of the new influences established among the program's blocks.

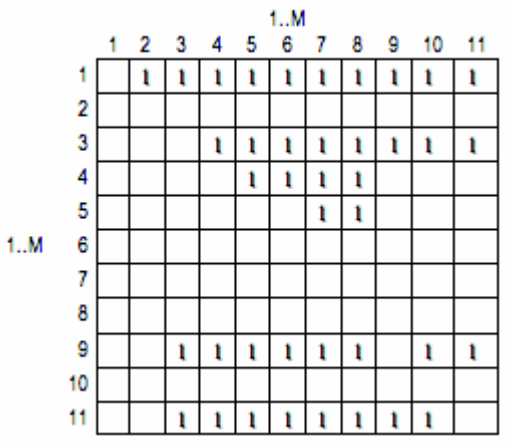

Figure 5. Matrix *IB* of the influences among the sample program's blocks.

5.3. Changes testing

After a program modification a very careful testing has to be made, in order to verify that the carried out changes do not degrade (instead to improve) software quality. In the maintenance phase it turns also out that it is usually impossible to get an exhaustive testing, and the testing optimization models aim to select a certain subset of paths to be tested (selective testing, [38]).

5.3.1. Finding the paths influenced by changes

We suppose to be useless to test those paths not interested, (i.e., whose blocks are not influenced) by any change because it means that they carry out their functionality correctly.

It is necessary to keep in mind, in the following exposition, that the runnable paths are extracted on the base of the detailed functional lists, and therefore to every path corresponds a program's functionality.

Starting with the matrices *A* (see 3.1) and *IB* (see 5.2), the paths influenced by the changes can be identified as those containing at least one block influenced by a change. Through these premises it is possible to develop an algorithm that allows to obtain the set of the paths (runnable before the modification) influenced by the change of the generic block b_k.

Let BJ the indexes set of the program's blocks influenced by the change. If J is the indexes set of the runnable paths before the change, and A_j is the jth column of the composition matrix A of the runnable paths, then:

$$PBJ = \{j \in J | A_j \cap BJ \neq \emptyset\}$$

is the indexes set of the paths influenced by the change.

5.3.2. Application of the testing optimization model

After making some changes to a program, one may consider to apply the testing optimization model to select one or more subsets of runnable paths. The testing of the chosen paths must guarantee that there has been no degradation of software quality after change application ([36], [34], [7]).

For the testing optimization model to be applied the runnable paths need to be defined, among those the solution model's algorithm must select the paths to test.

Let J^* the indexes set of all the paths not influenced by the changes; the paths in J^* are not therefore considered in the application of the model.

The paths influenced by the changes are represented by the set PBJ (see 5.3). Holding into account that every path identifies a program functionality, for each path in PBJ it can happen that:

- the functionality carried out by that path has been removed because no more required: the path is then no more runnable and will no longer be considered in model's application;
- the functionality carried out by that path is executed in a wrong man- ner: some changes have been adopted, the old path identified by a PBJ's index has been removed and a new one (not belonging to J^*) is defined, which must be considered in the model solution algorithm's application;
- new functionalities and then new paths have been introduced, which has to be considered in model's application.

Starting with the detailed lists of changes it is then possible to point out a set of runnable paths.

Let $J' = \{1, 2, \ldots, N'\}$ the set of runnable paths within the modified program. Usually, program's modifications imply the changes of some program's particular characteristics such as:

- total number of code lines;
- number of blocks M;
- matrices A, E, IB, W;
- values v_j for the paths in the set J^*.

Let us suppose the changes do not influence the paths identified by J^* : therefore such paths, being error-free and not modified, will not be considered in the model's application.

Let then c_j the test execution cost of the j-th path, and v_j its test execution value. Let M' the number of blocks and $W' = w'_{i,j} (i = 1 \ldots M', j = 1 \ldots N')$ the partial values matrix of the modified program.

The optimization model in its application to the modified program then becomes:

$$\max z(x) = \sum_{i=1}^{M} \max_{j=1 \ldots N} \{w_{i,j} x_i\}, \text{ subject to}$$

$$\sum_{j=1}^{N} c_j x_j \leq c_{\max}, \; x_j = \{0, 1\} \quad (j = 1 \ldots N).$$

In conclusion, it is opportune to consider the importance that covers the blocks effect of redundancy upon the cardinality of the set *PBJ*. It is easy to understand that the cardinality of such a set is proportional to the redundancy degree RE_i (see 3.3) of the blocks b_i modified or influenced by the changes.

In particular, if we suppose to modify an instruction of the block b_1 (the first block in the program, common to all the runnable paths; see figures 4 and 5) we have $RE_1 = 1$, i.e., the block belongs to all paths and so they all have been influenced by the change; therefore, card(PBJ) = N.

6. Conclusions

In order to solve (if ever possible to identify all the functional paths of a program) the problem of the optimal choice of paths for software testing (that is the identification of optimal test data) testing optimization models are definable, based on factors such as:

- the program structure (defining the path's test execution value, corre- lated with the number of executed instructions);
- the available resources for the program testing (defining the path's test execution cost, correlated with the execution time).

The defined software testing optimization models are the following bivalent integer linear programming models:

Model M_1

$$\max z_1(x) = \sum_{i=1}^{M} \max_{j=1..N}\{w_{i,j}x_j\}, \text{ subject to}$$

$$\sum_{j=1}^{N} c_j x_j \le c_{\max}$$

$$x_j = \{0, 1\} \quad (j = 1..N)$$

Model M_2

$$\min z_2(x) = \sum_{j=1}^{N} c_j x_j, \text{ subject to}$$

$$\sum_{i=1}^{M} \max_{j=1..N}\{w_{i,j}x_j\} \ge v_{\min}$$

$$x_j = \{0, 1\} \quad (j = 1..N).$$

Under particular conditions, such optimization models are also equivalents. The studied model (M_1) is that of maximization (see 3.4). The problems that can arise in the application of such optimization models are:

- the practical impossibility to identify all the runnable paths inside a program (or software system);
- a greater memory requirement to store the composition matrix of runnable paths, the execution matrix of runnable paths, the matrix of partial values, the representative tree of the "branch & bound" algorithm's application for the solution of the model;
- a greater execution time of the solution algorithm.

The first problem is of difficult (if not impossible) solution; however, the model gives an optimal solution with respect to all the paths identified in the program, and this at least guarantees the maximization of the program's topological coverage, even if some functionalities (and consequently some errors) may in effect escape to the testing.

As far as the other two points, a further optimization criterion of the solution algorithm consists in the possibility of reducing processing times and memory requirements through stop criteria based on a maximum allowed time (or steps' number) to come to a strictly feasible solution. As an example, processing could be stopped in case a new strictly feasible solution has not been generated in an equal interval to that needed for reaching the first strictly feasible solution. Of course, such a stop criterion could carry to a non-optimal solution; but it is important to remember that the algorithm gives, during processing, ever better strictly feasible (sub-optimal) solutions.

A real application of the testing optimization models is the optimal choice of paths to test after a program change during the software maintenance phase. We moreover think that the proposed model, together with the matrix of the influences between the blocks, can be a valid instrument for software testing during the maintenance phase. From a strictly practical point of view, it is just this last application that can turn out much effective to reach a high level of software quality.

The proposed testing optimization models M_1 and M_2 do not provide any suggestion (not even methodological) for estimating the number of residual errors in a program, after completing the testing activity. We think this is the direction toward which to address research, in order to achieve a deeper knowledge about software products reliability.

References

[1] N. Abbattista, O. Altamura, F. Esposito, A. Franich and G. Visaggio, Uno strumento di analisi per la manutenzione di un programma, Atti del Congresso AICA (1981), 651-655.

[2] V. Basili and R. Belby, Comparing the effectiveness of software strategies, IEEE Trans. on Soft. Eng. SE-13(12) (1987).

[3] M. Branstad, J. Ceriavskij and W. Adrion, Validation, verification and testing for the individual programmer, IEEE Computer 13(12) (1980), 24-30.

[4] R. Bryce and C. Colbourn, Prioritized interaction testing for pairwise coverage with seeding and avoids, Information and Software Technology Journal (IST, Elsevier) 48(10) (2006), 960-970.

[5] R. Bryce, A. Rajan and M. Heimdahl, Interaction testing in model-based development: Effect on model coverage, 13th Asia-Pacific Software Engineering Conf. Bangalore, India (2006).

[6] T. Cabe, A complexity measure, IEEE Trans. on Soft. Eng. SE-2(4) (1976), 308-320.

[7] M. Ceriani, D. Marini and L. Palmieri, Un esperimento di valutazione del testing di un programma in un ambiente industriale di produzione del software, Atti del Congresso AICA (1980), 831-840.

[8] M. Ceriani, A. Cicu and M. Maiocchi, Progettazione, produzione e manutenzione di tests per il controllo di qualita del software: un metodo con relativi strumenti, Atti del Congresso AICA 2 (1979), 26-35.

[9] L. Clarke, A system to generate test data and symbolically execute programs, IEEE Trans. on Soft. Eng. SE-2 (1976), 215-222.

[10] J. Darringer and J. King, Application of symbolic execute to program testing, IEEE Computer 11(4) (1978), 51-60.

[11] J. Duran and S. Ntafos, An evaluation of random testing, IEEE Trans. On Soft. Eng. SE-10(4) (1984).

[12] R. Fairley, Tutorial: Static analysis and dinamic testing of computer software, IEEE Computer 11(4) (1978), 14-23.

[13] G. Gannon, Error detection using path testing and static analysis, IEEE Computer 12(8) (1979), 26-31.

[14] D. Gelperin and B. Hetzel, The growth of software testing, Comm. of the ACM 31(6) (1988).

[15] T. Green, N. Schneidewind, G. Howard and R. Pariseau, Program structures, complexity and error characteristics, Proceedings of the Symposium on Computer Software Engineer, New York (1976), 139-154.

[16] M. Grindal, J. Offutt and S. Andler, Combination testing strategies: a survey, Software Testing, Verification and Reliability 15(3) (2005), 167-199.

[17] R. Hamlet, Special section on software testing, Comm. of the ACM, 31(6) (1988).

[18] W. Howden, An evaluation of the effectiveness of symbolic testing, E. F. Miller, W. E. Howden, Tutorial: Software Testing and Validation Techniques, IEEE, New York (1978), 300-313.

[19] W. Howden, Introduction to the theory of testing, E. F. Miller, W. E. Howden, Tutorial: Software Testing and Validation Techniques, IEEE, New York (1978), 16-19.

[20] W. Howden, Functional program testing, IEEE Trans. on Soft. Eng. SE-6(2) (1980), 162-169.

[21] W. Howden, Weak mutation testing and completeness of test sets, IEEE Trans. on Soft. Eng. SE-8(4) (1982).

[22] W. Howden, The theory and practices of functional testing, IEEE Software (1985).

[23] W. Howden, A functional approach to program testing and analysis, IEEE Trans. on Soft. Eng. SE-12(10) (1986).

[24] W. Howden, Reliability of the path analysis testing strategy, IEEE Trans. on Soft. Eng. SE-2(3) (1976), 208-214.

[25] T. Kuo-Chung, Program testing complexity and test criteria, IEEE Trans. on Soft. Eng. SE-6(6) (1980), 531-538.

[26] A. Land and A. Doig, An automatic method of solving discrete programming problems, Econometrika 28(3) (1960), 497-520.

[27] T. McCabe, Structured Testing, IEEE Computer Society Press, Silver Spring, Maryland (1982).

[28] E. Miller Jr., Program testing, IEEE Computer 11(4) (1978), 10-12.

[29] E. Miller and W. Howden, (Eds.) Tutorial: Software Testing and Validation Techniques, IEEE Computer Society Press, New York (1981).

[30] G. Myers, Software Reliability: Principles and Practices, John Wiley and Sons, New York (1976).

[31] G. Myers, A controlled experiment in program testing and code walk-throughs/inspections, Comm. of the ACM 21(98) (1978), 760-768.

[32] T. Ostrand and M. Balcer, The category-partition method for specifying and generating functional tests, Comm. of the ACM 31(6) (1988).

[33] C. Ramamoorthy, F. Siu-Bun and W. Chen, On the automated generation of program test data, IEEE Trans. on Soft. Eng. SE-2(4) (1976), 293-300.

[34] C. Ramamoorthy and F. Bastiani, Software reliability: Status and perspectives, IEEE Trans. on Soft. Eng. SE-8(4) (1982).

[35] N. Schneidewind, Application of program graphs and complexity analysis to software development and testing, IEEE Trans. on Reliability R-28(3) (1979), 192-198.

[36] A. Sorkowitz, Certification testing: A procedure to improve the quality of software testing, IEEE Computer 12(8) (1979), 20-24.

[37] P. Weide, Improving medical device safety with automated software testing, Med. Dev. Diag. Indust, 16(8) (6679), (1994).

[38] S. Zeil and L. White, Sufficient test sets for path analysis testing strategies, Proceedings of the 5s1 International Conference on Soft. Eng. IEEE, San Diego, California (1981), 184-191.

[39] S. Zeil, Testing for perturbations of program statement, IEEE Trans. on Soft. Eng. SE-9(3) (1983).

■